A Kullback-Leibler Divergence Exploration into a Look-Ahead Simulation Optimization of the Extended Compact Genetic Algorithm

Nathan Vasquez
College of Science and Engineering
University of Minnesota - Twin Cities
Minneapolis, MN 55455
Nathan.J.Vasquez@gmail.com

***Abstract*—The Kullback-Leibler Divergence of gene distributions between successive generations of the Extended Compact Genetic Algorithm (ECGA) is explored. Therein, the fragility of the algorithm's dependability to the beginning generations' biasing is suggested. A novel approach within the scope of the ECGA for choosing a better bias by allowing the ECGA to simulate itself is presented. It is shown that, by simulating itself, the ECGA is able to use a smaller population and evaluate fewer fitness calls while maintaining the same ability to find optimal solutions.**

I. Introduction

Genetics and the theory of evolution provide a powerful framework for shaping how populations evolve. Leveraging these techniques has been studied extensively in the efforts to make generalized optimization algorithms. The class of algorithms that models its optimization problems in such genetic terms are called genetic algorithms [5]. We restrict our attention here to the Extended Compact Genetic Algorithm (ECGA) [9] which falls under the Estimation of Distribution Algorithms (EDA) class of algorithms [17], [18]. EDA's use a Marginal Product Model (MPM) to factor the problem into related components which allows it to sample better solutions more frequently [11], [17], [19]. MPM's are discussed in detail in the Background section.

These evolutionary algorithms are often studied against simplistically defined problems whose difficulty may be scaled in a controlled fashion [7]. Although studied against these algorithms, the ECGA is intended to be a "practical, real-world" problem solver [11]. The ECGA has been successful in areas including "forest management, quantum chemistry, stock trading, and also as an improved learning mechanism in learning classifier systems" [11]. The ECGA has also been shown to be effective in the nonlinear programming problem of binary working fluid power cycle optimization [19]. For this reason, the ECGA often is studied in terms of fitness evaluations of potential solutions instead of conventional time complexity bounds.

In this work, the focus is on controlled problems as the ECGA works on them. A constructed Kullback-Leibler divergence (KLD) between successive generations is used as a tool for exploration. KLD is chosen specifically for the fact that [4] points out that there is a need for investigation of this as the meaning of such a quantity is not known within an evolutionary context [4]. We show that our construction of this KLD does not necessarily behave uniformly within a class of problems. However, for a class of problems that exhibits uniform behavior with our construction we infer sensitivity of the algorithm's success from the construction. Based off of this, we hypothesize that the ECGA is susceptible to biasing within its early generations. We propose a novel approach of investing time into simulating future generations, which we refer to as look-ahead simulations, to control the biasing of early generations outside the simulation. Herein, we show that, by using look-ahead simulations, the ECGA's required population size may be reduced by up to 20.5% and the required number of fitness calls may be reduced by up to 3.2% while maintaining the same ability to find optimal solutions.

II. Background

The contributions presented in this work discuss KLD as a distance between probability distributions, modifications to the behavior of the MPM's sampling function in the ECGA, and various parameterizations of the ECGA. Prior to this discussion in the Contributions section, we present here essential information related to these topics.

We begin by giving an overview of the generic genetic algorithm. We then present a deceptive problem which is commonly explored in tandem with genetic algorithms and discuss why they are not able to cope with this problem class. We use this deceptive problem to motivate and guide our discussion of linkage learning and then discuss how the ECGA uses an MPM to tackles this issue. After discussion of the ECGA, an overview of KLD is given.

A. Genetic Algorithm

Genetic algorithms solve optimization problems by evolving a given non-optimal solution. As its name suggests, genetic algorithms are rooted in genetics and draw on concepts from evolutionary natural selection. At a high overview, the algorithm takes as input members of a population (potential

solutions) and an environment (the problem). It then modifies the population by iterating through multiple generations of selecting mating partners, breeding (also known as crossover), mutating the offspring, and imitating natural selection by replacing individuals in the population by considering each individual's fitness as defined by the environment.

Various implementations of genetic algorithms are able to handle rich object-oriented approaches to these genetic operators of breeding, mutation, and natural selection. However, these problems are most often studied as binary strings as this makes reasoning, discussing, and presenting the individuals easier. Restriction to binary strings is without loss of generality, as these concepts may still be applied to finite-dimensional spaces [11]. The abstraction of genetics is hence overlaid upon these binary strings. Each member of the population is, in fact, a binary string. We refer to each bit interchangeably as a gene and assume a prokaryotic organism thus referring to the entire bitstring interchangeably as a chromosome. This assumption of a prokaryotic organism is only for the sake of uniform terminology and does not impose restrictions on the bitstring.

For later use we formally give two equivalent definitions of L that follow from our defined abstraction.

$$L := \text{Number of genes in an organism}$$
$$L := \text{Number of bits in the bitstring}$$

Note that L is referred to as the problem size.

We present one such genetic algorithm in Algorithm 1. Its purpose is solely to give visual aid of how such an algorithm would be laid out to achieve optimizing a problem.

Algorithm 1: Genetic Algorithm

```
1  Function GeneticAlgorithm(problem):
2      population = randomPopulation()
3      while not done do
4          newPop = []
5          while size(newPop) < size(population) do
6              parent1 = select(population)
7              parent2 = select(population)
8              offspring = breed(parent1,parent2)
9              if uniform(0,1) < chanceToMutate then
10                 offspring = mutate(offspring)
11             newPop.add(offspring)
12         replace(population, newPop)
13     solution = getBestIndividual(problem, population)
14     return (solution)
```

The various functions in the algorithm that are simulating natural phenomenon can be arbitrarily complex in their internal procedures. Although the implementations differ, [11] contends that it is widely agreed that genetic algorithms are successful because of their ability to hand off substructures of the problem to future generations [11]. Harik et al. discuss in depth a controversial topic without clear consensus: "the nature of the structures [a genetic algorithm] exchanges" [11]. They boil this down to two mutually exclusive options: either genetic algorithms can only effectively handle single genes at a time, or they are able to effectively handle larger structures within the chromosomes [11]. A note of clarification here may be needed. It may be wondered why we instead would not simply assume a eukaryotic organism and pass this structure in the form of multiple chromosomes in to the algorithm. The answer to this is that we wish to not have to pass this partitioning into the algorithm. We opt to use the same terminology as [19] and [11], and refer to these subsets of genes as building blocks.

Before continuing with the discussion of building blocks which will lead us to the concept of linkage learning, we detour with a discussion of a deceptive problem to motivate the need for linkage learning.

B. Deceptive Problems

In the study of genetic algorithms, three classes of problems are contrasted. These classes, as defined in [3], are those that give false information, no information, and correct information as to the distance a solution has to an optimal solution. These problems are referred to respectively as deceptive, hard, and easy problems. An example of an easy problem is the OneMax problem [7]. In this problem, an individual's fitness is simply the number of 1's appearing in its bitstring. We formally define OneMax similarly to [19] in Algorithm 3, but decompose this in terms of Unitation, found in Algorithm 2.

Algorithm 2: Unitation

```
1  Function Unitation(genes):
2      return (number of genes set to 1)
```

Algorithm 3: OneMaxFitness

```
1  Function OneMaxFitness(genes):
2      return (Unitiation(genes))
```

This problem of OneMax lends itself nicely to breeding functions. Assuming two individuals were selected because they had a high fitness, they'll both have an above average number of 1's in them. Thus when they combine to produce an offspring, they too will have an above average number of 1's. It is then up to future generations to combine different solutions to fill in more and more 1's. These genetic operators ultimately amount to a pseudo-hillclimbing effect.

Deceptive problems are introduced and studied with genetic algorithms to intentionally foil and punish this hillclimbing [18] with the intention that they will be difficult for the genetic algorithm to solve. A commonly studied such problem is the m-k deceptive trap problem [7], [11], [19]. In this problem there are m partitions of k bits, hence $L = mk$. The fitness of this problem is the sum of each fitness for the m partitions,

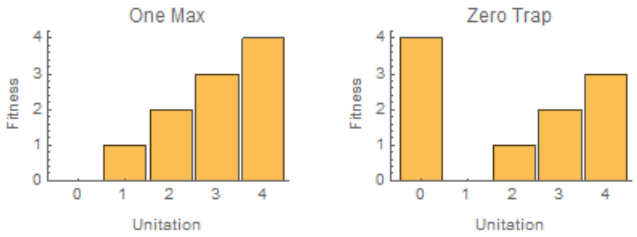
where each partition's fitness is defined as in Algorithm 4. We borrow from [7] the naming convention of prefixing problems with "one" or "zero" to reflect where that problem's maximum value is. For example, there are OneMax and its counterpart ZeroMax, OneTrap and it's counterpart ZeroTrap, etc.

Algorithm 4: ZeroTrapFitness

```
Function ZeroTrap(genes):
    u = Unitation(genes)
    if u = 0 then
        return k
    else
        return (u - 1)
```

Because both of these problems can be expressed in terms of unitation, we can view the fitness in terms of an individual's unitation as well. The graphs of a problem size 4 OneMax and a problem size 4 ZeroTrap with $m = 1, k = 4$ in terms of unitation are shown in Figure 1.

Fig. 1: Fitness in Terms of Unitation



(a) One Max (b) ZeroTrap-4

Throughout later discussion, we fix k, vary L, and leave m to be inferred; thus leaving us with the naming convention of trap-k. Because of this, we will only choose L to be multiples of each fixed k. For example, if we have a size 40 trap-4 problem, this problem has 10 subproblems ($m = 10$). To find the fitness of an individual in this problem, each of the subproblems will have their fitness calculated, and then the sum of those fitnesses will yield the fitness of the size 40 problem with 40 being the maximum fitness. Although it is irrelevant where these partitions are placed within the problem, we choose to place them adjacent to each other as in Figure 2. However, we mention again: we choose these partitions, but do not inform the genetic algorithm of such substructure within the problem.

Fig. 2: Gene Partitions in a Trap-4 Problem

$$\{[0 \,..\, 3], [4 \,..\, 7], [8 \,..\, 11], \ldots, [L-4 \,..\, L-1]\}$$

Considering Figure 1, and our previous discussion of hill climbing, we see how this trap function does its job in deceiving the genetic algorithm since it is rewarded by combining solutions to obtain more 1's in its population's bitstrings. With this in mind we continue in the next section by considering this problem of having our algorithm identify such substructures of the problem.

C. Linkage Learning

A simple genetic algorithm attempting to solve a size 40 ZeroTrap-4 problem will more likely than not produce a suboptimal solution consisting of some blocks of 0000, but mostly blocks of 1111 [11]. We consider though, that the genes making up these blocks are related (after all, we're running them through the same fitness function together), and it is precisely this correlation that a normal genetic algorithm fails to take advantage of [11]. The problem of deciding which genes are related is known as the linkage learning problem [11].

Defined more precisely, "Linkage learning in genetic algorithms is the identification of building blocks to be conserved under crossover." [11]. Harik et al. argue that this linkage learning problem of identifying which genes are linked is equivalent to that of learning probability distributions over multi-variate spaces [11]. It should be noted that this equivalence is under the assumption that linking genes in a population does, in fact, have this population available to it. In the case of learning a probability distribution, we assume individuals are tested against a fitness function by sampling such an individual from said probability distribution.

To ground ourselves in this concept we consider again our size 40 ZeroTrap-4 problem. Suppose our genetic algorithm gets lucky at some point and finds very nice potential solutions to one of these trap subproblems. Perhaps four members of the population have 0000 for its first subproblem, and the other six have have 1111. Because our breeding function does not know about this substructure, it is likely that these will combine into a binary string with 0's and 1's. Indeed, this combination has a lower fitness than either parent. Considering the order-1 probabilities of each individual gene without any further information doesn't do much better. We see that the probability for these bits is as follows: $P(\text{bit 1 is a 1}) = P(\text{bit 2 is a 1}) = P(\text{bit 3 is a 1}) = P(\text{bit 4 is a 1}) = 6/10$. Now we could construct individuals from these probabilities by sampling these probabilities with a random uniform variable from 0 to 1. However, by doing so we wouldn't be in much better or worse of a situation than we were with the population. These both suffer from the same issue in that they are treating each gene as independent.

Population-Based Incremental Learning (PBIL) [1] and the Compact Genetic Algorithm (cGA) [8] consider these order-1 probabilities to sample offspring. However, breaking up these fitness boosting substructures of our individuals is precisely our problem and limiting factor in these algorithms [11]. Consider instead a breeding function that works on these entire blocks as a unit. An offspring receiving a guaranteed block of 1111 or 0000 would be much better in terms of fitness than the fitness that the offspring would receive in breaking this block up. With our same example, this population's first building block would be $P(0000) = 4/10$ and $P(1111) = 6/10$. Of

course now, inherently, we've set all other combinations (for example 0001) to probability 0. Harik et al. argue that both the approach of sampling the building blocks from the population and sampling the building blocks' marginal probabilities achieve the same result of allowing the crossover operator to preserve this building block's fitness-boosting configurations across generations [11].

D. Extended Compact Genetic Algorithm

1) Marginal Product Model: Before presenting the algorithm, we discuss the machinery that the ECGA will use to tackle the problem of linkage learning. In the previous section, we saw that a potential schema is one that is able to represent the particular marginal probabilities that a set of genes has. A Marginal Product Model (MPM) is a class of probability model that does just this. As its name suggests it is a product of marginal distributions for a given set of genes. For concreteness we give an example of what a MPM might look like for a small arbitrary problem in Figure 3.

Fig. 3: Explicit Marginal Product Model

Gene Partitions	[0,3,5]	[1,2]	[4]
Marginal Probabilities	000 : 0.25 110 : 0.5 111 : 0.25	11 : 1.0	0 : 0.5 1 : 0.5

Something that may not be immediately apparent that we will investigate later is that we do not capture these probabilities precisely. Theoretically, such a probability model would exist to describe exactly what it means for a population to have its fitness boosted by certain configurations of genes. However, instead of capturing this precisely, the MPM may simply note the partitions and let sampling a current population define the marginal probabilities at these partitions. After all, that population is a finite sample from the theoretical model [11]. Hence, assuming we provide the MPM with a concrete population from which to sample with the same marginal probabilities for those partitions, it is sufficient to only capture the partitions. Again for concreteness, for the MPM given in Figure 3, we give an example in Figure 4 that describes the same information. Although, for complex models, it may be impossible to capture this precisely with a finite population [11].

Fig. 4: Implicit Marginal Product Model

Gene Partitions	[0,3,5]	[1,2]	[4]
Genes			
Individuals	011000 111100 111110 111111		

2) Combined Complexity Criterion: Lastly, all that is left to decide is what genes should be placed together in a partition. Prior to doing this however, we give discussion to a thought the reader may be wondering from a previous discussion: If the order-1 marginal probabilities on individual genes are not sufficient, why is it that we do not look to higher order behavior? In fact, some algorithms have delved into this approach with order-2 behavior [6] which have "sometimes been found to be vastly superior" [11]. However, although higher order behavior has the ability to model more complex and precise behavior, Harik et al. mentions, "the validity of doing so has been questioned [14]" [11].

It is the author's opinion that Harik does a phenomenal job leading the discussion to the rationale of how he chooses the criteria for gene partitions. Hence the following is an excerpt from [9] with our same definition of L:

> Pursuing this last train of thought to its ultimate conclusion reveals the flaw in its prescription. We can directly model the order-L behavior of polling the population, by only generating new members through random selection of chromosomes that exist in the population already. This behavior will rapidly lead to the algorithm's convergence, while exploring no new structures. Thus, more accurate modeling of the population's distribution is not always a desirable course of action . . . It is well known that unbiased search for such models is futile. Thus we have no choice but to select a bias in this search space. The one we choose is that given all other things are equal, simpler distributions are better than complex ones. Simplicity here can be defined in terms of the representational complexity of the distribution, given the original problem encoding. All things are, however, rarely equal, and there remains a tradeoff between simplicity and accuracy. Our aim will therefore be to find a simple model that nonetheless is good at explaining the current population . . . Motivated by the above requirement, we venture forth a hypothesis on the nature of good distributions: By reliance on Occam's Razor, good distributions are those under which the representation of the distribution using the current encoding, along with the representation of the population compressed under that distribution, is minimal.

It is with this Minimum Description Length (MDL) bias in the search for a good distribution that our problem is now phrased as a constrained optimization problem with the objective of minimizing the combined complexity [11]. The combined complexity criterion is defined in [11], as follows: Let

N := Population size,
S_i := Size of the ith partition of an MPM,
M_i := Marginal distribution over ith partition of an MPM,
p_k := Probability of outcome k,

and

$$Entropy(M_i) := -\sum_k (p_k \, log_2(p_k)).$$

Then, for a given partitioning of an MPM,

$$C_m := log_2(N+1) \sum_i 2^{S_i} - 1,$$

$$C_p := N \sum_i Entropy(M_i),$$

and finally,

$$C_C := C_m + C_p.$$

C_m we refer to as the model complexity, which represents the number of bits required to represent all of the marginal probabilities. C_p we refer to as the compressed population complexity which is the average number of bits it takes to represent a structure sampled from a partition's distribution, summed over all partitions, that value in turn multiplied by the population size [11]; A more convenient way of interpreting this is that C_p is the expected number of bits necessary to transmit the population compressed under the given partitioning as a message, although we note that the transmission is inapplicable to our current discussion.

At this point we have a problem defined to find the minimum value of C_C and simply try all possibilities to find the minimum. In fact, we can do much better with a greedy search implementing a cache [11], the implementation of which can be found in [16].

3) Extended Compact Genetic Algorithm Pseudocode: With the main component, the MPM, fleshed out, we present a high level parameterization of the Extended Compact Genetic Algorithm shown in Algorithm 5.

Algorithm 5: Extended Compact Genetic Algorithm

```
1 Function ECGA(problem):
2     population = randomPopulation()
3     while not done do
4         selectedSet = select(population)
5         MPM.generateModel(selectedSet)
6         newIndividuals = MPM.sample(selectedSet)
7         replace(population, newIndividuals)
8     solution = getBestIndividual(problem, population)
9     return (solution)
```

With a normal genetic algorithm, we had operators of selection, breeding, mutating, and replacement. In the case of the ECGA, we see that breeding and mutating have been removed, and now in their stead have the MPM.

Because the ECGA's main use is in complex problems for which often the fitness calculations are relatively expensive, much of the literature's [11], [19], [17] analysis of the algorithm has been in time units of fitness calls instead of conventional time complexity bounds. In the case of a size 40 trap-4 problem, the ECGA has been shown to be up to 1000 times faster than a regular genetic algorithm [11]. However, the parameterizations of this algorithm affect results such as these. As such, we discuss briefly the various forms of these parameterizations.

4) Selection: From our pseudocode abstraction above, it may seem as though we should be able to build our model from the population instead of selecting a representative set. However, it is important to understand that the selection in this algorithm plays the role of correlating genes with high fitness. Harik et al. comment on the sensitivity to selection that the ECGA has. Indeed, the MPM's greedy search for correlated genes starts with the assumption that all genes are independent and begins to merge those that are correlated. If selection fails to correlate patterns in the genes with high fitness, the ECGA will fail [11].

Several different approaches have been suggested: tournament selection with replacement, tournament selection without replacement, and truncation selection. Furthermore these parameterizations have been further parameterized by such things as choosing a selected set with size given a proportion to the original population [16]. We restrict ourselves to the most common approach of tournaments without replacement and choose 16 as our tournament size. As is more typically done, we choose our selected set to be the same size as the original population.

5) Replacement: A straightforward replacement method is that of full replacement, where we take the old population and replace all individuals with those generated from our MPM sampling. In this context of full replacement, sampling produces a population with size equal to the original population. However, this equality is not a strict requirement; We could conceivably model a population with the birth rate not equal to the death rate, but this hasn't been studied within the scope of the ECGA to the best of our knowledge.

If we choose replacement methods other than full replacement, we remove the restriction that the sampled population must be the same size as the original population and call the proportion of sampled individuals the $offspring\ ratio$ (this ratio is referred to in [16] as offspring size). An example of this would be worst replacement [16]. For example if we have a population of size 1000, and we choose the offspring ratio to be $1/2$, this would cause our MPM to produce 500 individuals, which in turn would replace the 500 individuals in the current population with the lowest fitness.

Lastly we consider restricted replacement (explained in [10] where they refer to this as restricted tournament selection) which needs a tournament size, which we will call the window size. We leave this window size as $N/20$ as recommended in [15]. This approach uses a distance metric over the population, taking two individuals and mapping them to a distance (we focus within the scope of hamming distance). For each individual this replacement method seeks to replace into the population, it considers random individuals in the population of size equal to the window size. It then computes the distances for each of these members. It chooses whichever individual has

the closest distance to the individual seeking to be replaced into the population. With these two individuals, it keeps in the population the most fit individual of the two. What this selection methods allows for us to do is to create localized niches in the population.

In [18], the authors argue extensively the requirement of EDA's to have a sub-structural niching method when applied to multimodal, hierarchical, dynamic, and multiobjective optimization problems. [18] provides a more advanced technique that performs on par with restricted replacement, but does so with less population. Within the scope of this paper, we focus our attention on the restricted replacement.

E. Kullback-Leibler Divergence

In the previous section, we discussed extensively the ECGA. Now we seek to describe the behavior of the population changing with respect to the modifications the algorithm makes to the population through replacement. We use Kullback-Leibler Divergence (KLD) as our statistic to do just this. Before our explanation of this quantity, it should be noted that [4] points out that "the evolutionary meaning of this quantity is not known and needs to be further investigated" [4]. Later, we scratch the surface of this within the scope of the ECGA, but for now we present this concept in more general terms.

We opt to use more conventional notation, but nonetheless, [13] defines KLD as follows:

Given P, Q as discrete probability distributions, the KLD from Q to P is defined to be

$$D_{KL}(P||Q) := \sum_i P(i)\, log_2 \frac{P(i)}{Q(i)}$$

Note that this definition can be defined for any base logarithm, but we choose ours to be base 2 for units of bits. We note further a few restrictions, properties, and an interpretation of this quantity and leave further discussion for later.

1) Restrictions of KLD from Q to P:

- $Q(i) = 0$ must imply $P(i) = 0$, or else KLD is not defined.
- Contribution of the ith term is defined to be zero in the event that $P(i)$ is 0.

2) Properties of KLD from Q to P:

- $D_{KL}(P||Q)$ is nonnegative
- KLD is a non-symmetric difference between two probability distributions, i.e. it does not obey the triangle inequality ($D_{KL}(P||Q)$ need not equal $D_{KL}(Q||P)$.)

3) Interpretation of KLD from Q to P:

- In Bayesian inference, when one revises one's beliefs from prior probability distribution Q to posterior probability distribution P, this quantity measures information gain [2].

III. Contributions

Discussed in the background section, we built up the rationale of the components of the ECGA which allows it to learn linkage within the set of genes in a chromosome. As the ECGA runs, it revises its beliefs as to what the structure of the problem is in the form of gene partitions. As the ECGA converges on a solution, it decreases the entropies of these sets of genes' marginal distributions.

With this in mind, our contributions are twofold:

1) A construction of KLD is imposed over the genes' order-1 probabilities and is shown not to exhibit uniform behavior across different classes of problems, nor does it necessarily exhibit uniform behavior within the same class of problem. Using this construction for a problem in the deceptive class of problems which exhibits uniform behavior, a correlation with finding an optimal solution is shown.
2) It is further shown that in this deceptive class of problems, look-ahead simulations are an effective method of influencing the entropies of the ECGA's gene partitions' marginal distributions when they're at their most sensitive state. This makes the ECGA more reliable, and thus allows for higher dimensional problem sizes to able to be solved with less population and fewer fitness calls.

All results that follow were obtained using the framework described in [16] and available via a public repository.

A. Kullback-Leibler Divergence Exploration

Given a marginal distribution of a partition of genes within an MPM, it would be ideal to consider how these marginal distributions change from generation to generation. Unfortunately, from generation to generation the MPM is constantly updating its partitioning.

Attempting to consider the partitions that remain the same would necessitate the interpretation of an aggregate collection of variably existent statistics. Instead of attempting to impose meaning onto such statistics, we ignore the boundaries of the partitioning and focus on the order-1 probabilities of the genes. We construct another probability distribution by scaling these genes' probabilities to sum to 1. We refer to this constructed distribution at generation i G_i.

Now, let us consider $D_{KL}(G_{i+1}||G_i)$, i.e. the KLD from one generation to its successor under our construction. We note that this statistic is particularly fickle with respect to our mirrored problems with max fitness at all 1's versus its counterpart with max fitness at all 0's. We consider a few examples to illustrate this point. In the following examples, let us restrict ourselves to an arbitrary size 4 problem and assume a population of size 8. We show the gene frequency counts pertaining to how many individuals have a gene of 1 in that gene's index and call this collection F_n for generation n.

In all of these examples, we start with four individuals of the eight having gene k set to 0 and the other four having gene k set to 1. Thus our initial configuration of F_i is $[4, 4, 4, 4]$ in every example. Note that it does not matter which individuals have a particular gene configuration. In Example 1, we see that, in the posterior generation, the arbitrary problem suggested through the fitness of these individuals that it was best to set all individuals' first and second genes to 0 and the

Example 1

Let
$F_i := [4, 4, 4, 4]$ *and*
$F_{i+1} := [0, 0, 4, 4]$.
Then
$G_i := [1/4, 1/4, 1/4, 1/4]$,
$G_{i+1} := [0, 0, 1/2, 1/2]$, *and*
$D_{KL}(G_{i+1}||G_i) := 1.0$.

Example 2

Let
$F_i := [4, 4, 4, 4]$ *and*
$F_{i+1} := [8, 8, 4, 4]$.
Then
$G_i := [1/4, 1/4, 1/4, 1/4]$,
$G_{i+1} := [1/3, 1/3, 1/6, 1/6]$, *and*
$D_{KL}(G_{i+1}||G_i) := 0.0817$.

algorithm stochastically did just this. In Example 2, we see that in the posterior generation, the arbitrary problem suggested it instead best to set all individuals' first and second genes to 1. Note that the KLD is drastically different in these examples. In Example 3, we note another issue that can arise: although the gene frequencies changed drastically from half set to 1's to all set to 1's, there's no difference in our constructed distribution, hence the KLD is zero.

From these examples we conclude that the KLD will not be useful in describing arbitrary classes of problems. This is exemplified by simply comparing OneMax against ZeroMax where we recognize that their genes will tend to either all 1's or all 0's respectively in a near uniform fashion, which will obtain drastically different results in terms of KLD. Another issue arises when there is a global shift in the genes' marginal probabilities, i.e. a uniform trend in the same direction (as in Example 3). This global shift will not be detected by our construction.

In light of these complications, it seems that our construction may not be useful to us. However, [19] explores the type of behavior found in Example 3, and coin the terms flash identification and sequential identification to describe the manner in which the ECGA discovers the building blocks within the problem. Flash identification is when all building blocks are discovered all at once, whereas sequential identification is when building blocks are discovered over the span of multiple generations. Particularly of note from their findings is that, as the difficulty of the problem goes up, this behavior changes from flash identification to sequential [19].

Example 3

Let
$F_i := [4, 4, 4, 4]$ *and*
$F_{i+1} := [8, 8, 8, 8]$.
Then
$G_i := [1/4, 1/4, 1/4, 1/4]$,
$G_{i+1} := [1/4, 1/4, 1/4, 1/4]$, *and*
$D_{KL}(G_{i+1}||G_i) := 0.0$.

For the first few generations of the ECGA with a randomized population, it is a fair assumption that our population is split roughly down the middle of having a 1 or a 0 for gene i. Furthermore, for difficult problems, given that they are sequentially learned, we should expect within the first few generations that the MPM will identify partitions incorrectly. Given that the MPM identifies building blocks incorrectly, we expect genes that are incorrectly linked to other genes not to stray far from previous generation's.

With this in mind, we consider Examples 4, 5, 6, and 7 through the lens of the situation just described and contrast with the previous examples.

Example 4

Let
$F_i := [4, 4, 4, 4]$ *and*
$F_{i+1} := [3, 3, 4, 4]$.
Then
$G_i := [0.25, 0.25, 0.25, 0.25]$,
$G_{i+1} := [0.21, 0.21, 0.29, 0.29]$, *and*
$D_{KL}(G_{i+1}||G_i) := 0.015$.

Example 5

Let
$F_i := [4, 4, 4, 4]$ *and*
$F_{i+1} := [2, 2, 4, 4]$.
Then
$G_i := [0.25, 0.25, 0.25, 0.25]$,
$G_{i+1} := [0.17, 0.17, 0.33, 0.33]$, *and*
$D_{KL}(G_{i+1}||G_i) := 0.082$.

In Examples 4 and 6, we have a small change in the frequencies of genes one and two. As Examples 4 and 6 get closer to Examples 1 and 2, we expect to see something like

Example 6

Let

$F_i := [4, 4, 4, 4]$ *and*

$F_{i+1} := [5, 5, 4, 4]$.

Then

$G_i := [0.25, 0.25, 0.25, 0.25]$,

$G_{i+1} := [0.28, 0.28, 0.22, 0.22]$, *and*

$D_{KL}(G_{i+1}||G_i) := 0.009$.

Example 7

Let

$F_i := [4, 4, 4, 4]$ *and*

$F_{i+1} := [6, 6, 4, 4]$.

Then

$G_i := [0.25, 0.25, 0.25, 0.25]$,

$G_{i+1} := [0.3, 0.3, 0.2, 0.2]$, *and*

$D_{KL}(G_{i+1}||G_i) := 0.029$.

Examples 5 and 7 respectively. This suggests that Examples 5 and 7 should be less likely to occur than 4 and 6. Given these likelihoods, we consider now the KLD of these examples. For these we introduce a temporary notation that is easier to follow. Let $\{k\}$ be the equivalence class of taking $F_i = [4, 4, 4, 4]$ to $F_{i+1} = [k, k, 4, 4]$. Hence $\{5\}$ represents the situation in Example 6 whereas $\{2\}$ represents Example 5. Furthermore, let $D\{k\}$ be the KLD calculated in this situation, for example $\{5\} = 0.009$. Note that $\{4\}$ is equal to the KLD calculated in Example 3.

With our more convenient notation, we outline an important point. Consider $\{0\}$, $\{4\}$, and $\{8\}$. In order, these represent the extreme case of setting the first two genes to all 0's, the case where there is no change to the population, and the extreme case of setting the first two genes to all 1's. Note that $D\{4\} < D\{0\} < D\{8\}$. However in the discussion to follow we aren't particularly interested in the whether $D\{i\} < D\{j\}$ for $i < 4$ and $j > 4$. Instead we are interested only in the behavior of those values of $D\{i\} < D\{j\}$ for the case where $i, j < 4$ or in the case $i, j > 4$.

From our examples, we see that

$$D\{4\} < D\{5\} < D\{6\} < ... < D\{8\}$$

and

$$D\{4\} < D\{3\} < D\{2\} < ... < D\{0\}.$$

As OneMax and ZeroMax have shown us, our construction will not be applicable in cases involving flash identification. However, for more difficult problems which we assume to be sequentially learned by the argument in [19], we should be able to see such trends for early generations. We hence focus on Trap-4 problems. Because OneTrap and ZeroTrap tend towards the specific poles of all 1's or all 0's we should see two types of behaviors. In the case of OneTrap, we should see behaviors like $\{i\}$ for $i > 4$ and in the case of ZeroTrap, we should see behaviors like $\{i\}$ for $i < 4$ with $\{i\}$ in either case being more likely with values closer to 4.

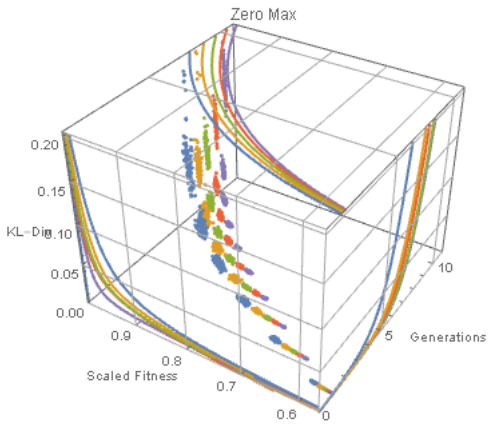
From the previous discussion, generalizations of the behavior in regard to changes that are not as controlled as those above are especially difficult because, as we additively overlap such examples, this behavior devolves into a global shift, and thus our statistic vanishes. Beyond the first few generations, the assumptions upon which this predicted behavior relies vanish as well, since the genes will no longer be close to uniformly distributed.

1) Corroborating Evidence: In the previous section, we noted the difficulty in generalizations and aim to corroborate claims based on empirical evidence. [19] gives an equation for suggested population sizes. In this experimental setup and others to follow, we use the term tune a population size to be the following procedure. We consider the suggested sizes (on the order of 1000-4000) in [19] and halve these. We manually observe when the algorithm approximately reaches the desired success rate. We then run at least 30 samples each containing 100 trials of the ECGA running from start to finish. These samples are taken in uniform increments, spread across an interval centered on our manually observed estimate. The radius of this interval was chosen to be at least 1000. A best fit line is fit to this data, and from this line we obtain our population size yielding our desired success rate.

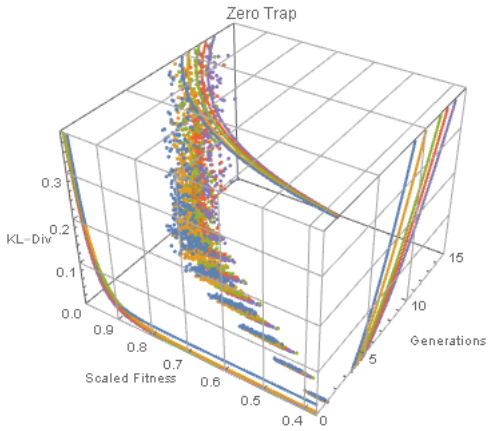
For the first experiment, we note that the equation given in [19] for population size is linear in the problem size. We tune a population size for a size 80 OneTrap-4 to have a 50% success rate, where success is finding an optimal solution, and then tune the same for a size 160. We then interpolate population sizes linearly to the other problem sizes. Note that we do not make efforts to cause OneMax and ZeroMax to fail since these are easy problems. We use a tournament size of 16 and selection method of tournament selection without replacement and refer to these two in the future as the standard setup. Furthermore, we use full replacement, and halt upon population convergence to a single individual (all individuals are the same). For problems among $\{\text{Zero}, \text{One}\} \times \{\text{Max}, \text{Trap-4}\}$, we vary the population from 80 to 160 in steps of 20. With this setup, we ran 100 trials for each variation and obtained the graphs in Figure 5.

Figure 5 demonstrates several behaviors of the problems under the constructed KLD. For the Trap problems, we see that within the first few generations the KLD follows anticipated behavior given the assumption that the ECGA sequentially identifies building blocks. OneMax exemplifies the necessity of this assumption given that OneMax exhibits flash identification. However, by the nature of our construction, ZeroMax ends up showing a trend in the graph similar to that of the Trap problems. Exploration into why ZeroMax exhibits this

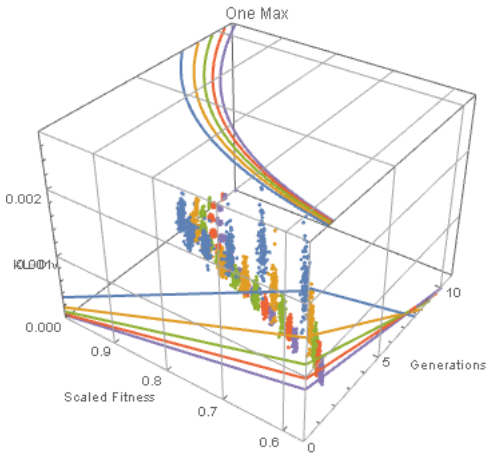
Fig. 5



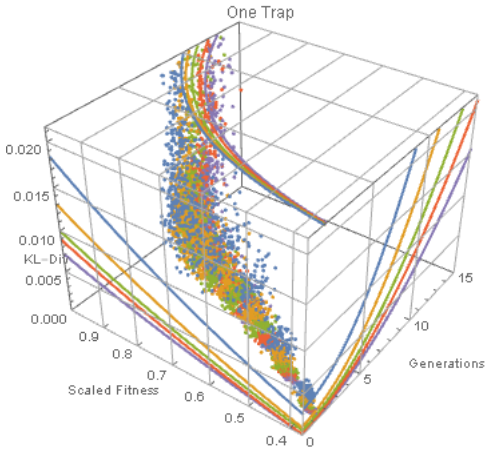
(a) Zero Max



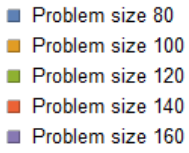
(b) ZeroTrap-4



(c) One Max



(d) OneTrap-4



(e) Legend Pertaining to (a)-(d)

(a) ZeroMax. (b) ZeroTrap-4. (c) OneMax. (d) OneTrap-4. Because the optimal solution is equal to the problem size which we are varying, we define Scaled Fitness to be fitness divided by optimal fitness.
For a better grasp of the data, we project the data into each plane and show trend lines on that plane's surface.

behavior without qualification of our assumption will not be explored here. The main corroboration we are interested in is the Trap problems as representatives of the deceptive class of problem.

The rest of our discussion is confined to improving the performance of the ECGA with difficult classes of problems, with Trap-4 as a representative. Hence, we focus our attention in the next section to the correlation of KLD under our construction with finding an optimal solution.

2) Correlation with Success: Within the previous results from Figure 5, we observe the correlation of our constructed KLD with the success of the algorithm to find the optimal solution in the following manner. We sum our constructed KLD over the first k generations for k from 1 to 3 and calculate the Pearson Correlation with the success of finding an optimal solution.

In the discussion to follow, we let C_k refer to the column in Table I pertaining to the Pearson correlation of $\sum_{i=1}^{k} D_{KL}(G_{i+1}||G_i)$ with finding an optimal solution. For instance, C_3 of $\text{ZeroTrap}_{\text{problem size 80}}$ is -0.26220. We remind ourselves that this is the sum of our constructed KLD going from the first to second, second to third, and third to fourth generations. In interpreting this data, we note that in a Bayesian inference view of the situation, $\sum_{i=1}^{3} D_{KL}(G_{i+1}||G_i)$ is the total bits of information by which the algorithm has revised its beliefs in the first three belief revisions.

Among the C_1 correlations, we see that there is little

TABLE I: Correlation with Finding Optimal Solution

Problem	Problem Size	Pearson Correlation of $\sum_{i=1}^{1} D_{KL}(G_{i+1}\|\|G_i)$ with Finding Optimal	Pearson Correlation of $\sum_{i=1}^{2} D_{KL}(G_{i+1}\|\|G_i)$ with Finding Optimal	Pearson Correlation of $\sum_{i=1}^{3} D_{KL}(G_{i+1}\|\|G_i)$ with Finding Optimal
ZeroTrap	80	0.11783	-0.02264	-0.26220
ZeroTrap	100	-0.20604	-0.16814	-0.31261
ZeroTrap	120	0.04840	-0.08642	-0.27657
ZeroTrap	140	0.00998	0.00136	-0.20139
ZeroTrap	160	0.05146	-0.15555	-0.23681
OneTrap	80	-0.10253	-0.19753	-0.28662
OneTrap	100	0.10800	0.07462	-0.23664
OneTrap	120	0.07597	-0.03784	-0.23690
OneTrap	140	-0.02730	-0.02054	-0.19642
OneTrap	160	-0.17927	-0.19902	-0.31136
Average		-0.01035	-0.08117	-0.25575

association. This means that, regardless of whether the KLD is large or small, there is negligible association with finding an optimal solution. In the context of our constructed KLD, having a larger KLD means there was a larger revision to the population and hence a larger magnitude of entropy loss. According to C_1 this entropy loss is irrelevant, i.e., the ECGA can recover from this.

Next considering C_2, we note that having a large value for $\sum_{i=1}^{2} D_{KL}(G_{i+1}||G_i)$ means that there was a large change in beliefs twice in a row. Note that C_2 is now more negatively correlated with success than C_1. Again, we see this trend in C_3 as well. This trend suggests that the ECGA cannot handle repeated large entropy losses. However, at the same time, the ECGA must reduce the entropy of its population since it requires this entropy loss to converge on potential solutions.

These correlations, of course, do not mean causation, but we speculate that the beginning generations are most important due to the fact that there is the most entropy in the system. Based on this assumption, we endeavour to make improvements on the algorithm.

B. Look-Ahead Simulations

As we noted in the previous section, we make the claim that the entropy loss in the system during the first few generations is more influential to the ECGA's ability to find an optimal solution than the entropy loss later in the system.

This guides our aim at improving this algorithm. If our improvement adds time proportional to the fitness evaluations in a given generation, our algorithm will still run in the same time complexity bound. However, in terms of raw fitness evaluations, assuming a proportion of 1 and the algorithm running the same number of generations, we will have doubled our fitness calls. Given this approach, we will impose on ourselves the requirement to find the solution twice as fast in order for this approach to be worthwhile. Finding a solution twice as fast seems incredibly unlikely especially as [19] notes that for large problem sizes, convergence is proportional to the square root of the problem size. Hence we focus our attention on the beginning of the algorithm's entropy loss.

Consider a run of the algorithm from start to finish without finding an optimal solution. At some point during its run, the MPM must have partitioned genes such that, when sampled, this produced an entropy loss in the system. We point out again that [19] finds that the algorithm only finds an optimal solution when all building blocks are correctly identified. Additionally, for difficult problems and especially for deceptive ones, this entropy loss is biasing our population towards a suboptimal solution. By the time the building blocks are discovered, there isn't enough entropy left in the system for the algorithm to do anything useful. We therefore conclude the following: If early generations had future generations' refined beliefs of what the problem looks like, access to those beliefs could allow the MPM to reduce the entropy of the population in such a fashion that the system would be biased more towards an optimal solution. To avoid evaluating too many fitness calls, we endeavour only to have the beginning few generations be informed of such information.

The only issue left is that, on a given generation, the MPM needs to come back to the current population its sample function was called with after considering these future generations. At the same time, it also needs to modify this population to consider future generations. Hence we have the algorithm simulate itself running. That is to say, in the first few generations, upon calling the MPM to sample individuals, we have the MPM make a copy of the algorithm and population, MPM included. We have the MPM's sampling function then simulate the algorithm running into the future to find a refined solution to where the simulated MPM suggests the partitions are and then halts the simulation. By having the MPM return to its entropy-rich, original population, the MPM can impose

TABLE II: Look-Ahead Simulation Parameterization

Parameter	Description
$generations\ to\ simulation$	Number of generations to use a modified MPM sampling technique that simulates up to a future MPM and instead uses its partition for sampling outside the simulation. After this generation, the algorithm returns to normal behavior.
$simulation\ generation\ depth$	Number of generations into the future a simulation should simulate for.
$simulation\ offspring\ ratio$	Same as normal offspring ratio except this size is used only during the simulation.

TABLE III: Size 80 OneTrap-4 Tuned to 50% Success Rate

$generations\ to\ simulate$	$simulation\ generation\ depth$	$simulation\ offspring\ ratio$	$offspring\ ratio$	Trials That Found Optimal Solution	Mean Fitness Evaluations Until Optimum found	STD Fitness Evaluations Until Optimum Found
0	N/A	N/A	0.5	215	15414.5	4814.30
1	1	0.5	0.5	722	17682.8	2279.50
1	1	1	0.5	726	17823.4	2370.81
1	2	0.5	0.5	762	14986.5	3252.87
1	2	1	0.5	737	15045.4	3374.42
2	1	0.5	0.5	680	17966.9	2378.30
2	1	1	0.5	666	18241.9	2476.46
2	2	0.5	0.5	654	15633.4	3400.24
2	2	1	0.5	657	15615.4	3374.97
0	N/A	N/A	1	486	17669.3	2771.34
1	1	0.5	1	728	17831.8	2465.26
1	1	1	1	734	17929.3	2630.98
1	2	0.5	1	763	14959.4	3410.10
1	2	1	1	765	14929.0	3187.57
2	1	0.5	1	657	18055.4	2407.33
2	1	1	1	687	18098.9	2623.29
2	2	0.5	1	658	15697.0	3504.10
2	2	1	1	645	15816.2	3595.38

Note that trials that did not find an optimal solution are not factored into the mean or standard deviation.

a better bias into the population. We refer to this process of the MPM simulating up to a future MPM's gene partitions a look-ahead simulation.

Presented in Table II is a parameterization of the look-ahead simulation just described.

For clarity, we point out that there are two separate MPM sampling functions: a normal one and one that does simulations. When the look-ahead simulation's sampling function is running, it only calls the normal sampling function. I.e. there are no simulations inside simulations.

It cannot be beneficial for $generations\ to\ simulate$ or $simulation\ generation\ depth$ to be large since it will increase the number of fitness evaluations too much. We constrict these parameters to the set $\{0, 1, 2\}$, and leave higher values to be studied based on preliminary results. Note that $generations\ to\ simulate$ set to 0 is the normal algorithm. Based off of claims from [11], only generating half of the next population can be a viable option which seems like a straightforward, potentially viable way for the algorithm to further make up time for the expended fitness calls. Hence, we further vary $simulation\ offspring\ ratio$ and $offspring\ ratio$ for values in the set $\{0.5, 1.0\}$.

In the following experiments, we parameterize the algorithm to its best known configuration to the author's knowledge. We use a problem size of 80 on a OneTrap-4 problem with restricted replacement and the aforementioned standard setup. Because restricted replacement outperforms the full replacement used previously, we retune our population size to yield a 50% success rate. This population size was found to be 1899. The results of running 1000 trials for each configuration with this experimental setup are shown in Table III.

We remind ourselves that, in the setup, these results are based on a configuration of the ECGA that had its population size tuned to fail half of the time. Because of this, we should not infer that the normal algorithm cannot solve these problems. Instead, we note that the algorithm simply requires a higher population size. By considering only the entries with the same $offspring\ ratio$, we can see when look-ahead simulations might require less population than the normal ECGA. However, there is still the cost of the extra fitness calls to be considered. We note also that, this low of a population is not a typical parameter settings for the algorithm.

We opt to illustrate this point with the preliminary results of the experiment we just discussed. The following experiment is identical to the one previous, with the exception of the population size. Where the previous population size was set to 1899 to achieve a 50% success rate, the following experiment had its population size set to 2000, which is more towards the spectrum of the algorithm's suggested population size on the order of 4000. The results of running 1000 trials for each configuration with this experimental setup using 2000 as the population size are shown in Table IV.

Debatably, increasing $generations\ to\ simulate$ appears to be more appealing given its success rate with respect to this population size increase. However, there is still the trade off with the increased fitness calls. We leave hypothesis testing of inferred trends in these data as future work. Accepting these results as circumstantial evidence that look-ahead simulations may be beneficial, we focus our attention here on testing this theory.

Hence we use the previous parameterizations and tune the population such that the algorithm finds a solution 95% of the time. The results of running 1000 trials for each configuration with this experimental setup are shown in Table V. Included in Table V are the population sizes that the parameterizations tuned to.

Considering the results in Table V with an $offspring\ ratio$ of 0.5, we see that unanimously, there was not any parameterization that performed better in terms of fitness calls than the original algorithm. However, all of these decreased the

TABLE IV: Size 80 OneTrap-4 Untuned

$generations\ to\ simulate$	$simulation\ generation\ depth$	$simulation\ offspring\ ratio$	$offspring\ ratio$	Trials That Found Optimal Solution	Mean Fitness Evaluations Until Optimum found	STD Fitness Evaluations Until Optimum Found
0	N/A	N/A	0.5	326	16162.6	4926.03
1	1	0.5	0.5	511	15172.2	4400.72
1	1	1	0.5	604	15165.6	3859.35
1	2	0.5	0.5	686	14723.0	3640.28
1	2	1	0.5	840	15167.9	2965.70
2	1	0.5	0.5	548	15711.7	3911.84
2	1	1	0.5	700	16508.6	3446.91
2	2	0.5	0.5	762	15908.1	3125.41
2	2	1	0.5	887	18260.4	1965.52
0	N/A	N/A	1	575	17826.1	2803.25
1	1	0.5	1	748	17951.9	2581.54
1	1	1	1	789	18243.3	2371.92
1	2	0.5	1	843	17727.2	2305.95
1	2	1	1	925	18585.9	1813.38
2	1	0.5	1	773	18571.8	2365.35
2	1	1	1	837	19550.8	1994.94
2	2	0.5	1	870	19137.9	1910.98
2	2	1	1	948	22130.8	1504.58

Note that trials that did not find an optimal solution are not factored into the mean or standard deviation.

TABLE V: Size 80 OneTrap-4 Tuned to 95% Success Rate

$generations\ to\ simulate$	$simulation\ generation\ depth$	$simulation\ offspring\ ratio$	$offspring\ ratio$	Population Size	Trials That Found Optimal Solution	Mean Fitness Evaluations Until Optimum found	STD Fitness Evaluations Until Optimum Found
0	N/A	N/A	0.5	2976	958	15397.2	2672.59
1	1	0.5	0.5	2803	955	15890.2	2600.43
1	1	1	0.5	2680	954	16051.9	2246.94
1	2	0.5	0.5	2620	962	15675.1	2162.14
1	2	1	0.5	2292	935	16133.5	2330.72
2	1	0.5	0.5	2791	951	16955.2	2381.88
2	1	1	0.5	2544	938	17862.2	2018.83
2	2	0.5	0.5	2471	933	17396.9	2042.47
2	2	1	0.5	2332	951	20425.2	1429.29
0	N/A	N/A	1	2665	949	19660.3	2142.92
1	1	0.5	1	2446	940	19422.3	2087.11
1	1	1	1	2407	960	19682.2	1990.60
1	2	0.5	1	2327	946	19030.7	1948.83
1	2	1	1	2120	951	19347.5	1807.88
2	1	0.5	1	2481	959	20501.5	1967.13
2	1	1	1	2294	945	21209.2	1872.61
2	2	0.5	1	2355	969	21190.6	1603.89
2	2	1	1	2007	938	22087.7	1499.40

We use a shorthand notation to refer to entries in this table such as $\{2, 1, 1.0, 0.5\}$ to mean the entry at $generations\ to\ simulate = 2$, $simulation\ generation\ depth = 1$, $simulation\ offspring\ ratio = 1.0$, $offspring\ ratio = 0.5$. Note that trials that did not find an optimal solution are not factored into the mean or standard deviation.

required population size and variance of how many fitness evaluations were called before finding the optimum. Despite this tradeoff, the case of an $offspring\ ratio$ of 1.0 is able to do better.

Considering the results in Table V with an $offspring\ ratio$ of 1.0, we see that setting $generations\ to\ simulate$ to 2 becomes too expensive of a process in terms of fitness calls. Among $generations\ to\ simulate$ set to 1, with the exception of $\{1, 1, 1.0, 1.0\}$, all of those among $\{1, 1, 0.5, 1.0\}$, $\{1, 2, 0.5, 1.0\}$, and $\{1, 2, 1.0, 1.0\}$ outperformed the original algorithm in all metrics. Contrasting these settings, $\{1, 2, 0.5, 1.0\}$ reduces the fitness calls the most, but is beat in the other metrics. This setting of $\{1, 2, 0.5, 1.0\}$ reduces the required fitness calls by 3.2%. Conversely, $\{1, 2, 1.0, 1.0\}$ reduces the other metrics, but is beat by fitness calls. This setting of $\{1, 2, 1.0, 1.0\}$ reduces the required population size by 20.5%.

IV. Conclusion

Because KLD has not been explored in an evolutionary context [4] this work explored a construction of KLD over the ECGA's population's genes' order-1 probabilities. It was found that this construction of KLD was not applicable to all classes of problems. However, for the particular problem of Trap-4, it proved useful in aiding the study of the algorithm's behavior on the problem. It was shown that large successive

refinements to the population's genes were inversely correlated with the ECGA finding an optimal solution.

Within the scope of deceptive problems, our results suggest that the ECGA is especially susceptible to biasing in its first few generations in terms of finding better solutions occurring, and less so in later generations. We found it beneficial for the ECGA to spend extra time within the first few generations simulating up to future generations' gene partitions created by the MPM. This allows the ECGA with look-ahead simulations to achieve similar results to the normal algorithm, but with up to 20.5% smaller populations and up to 3.2% fewer fitness evaluations. Although maximizing one these reductions comes at the cost of decreasing the reduction of the other, in either case, both the population size and number of fitness evaluations are able to be made simultaneously smaller than the original algorithm while maintaining the same ability to find an optimal solution.

V. Future Work

A. Kullback-Leibler Divergence

We discussed different behaviors that various problems exhibited in terms of our constructed KLD. Because genetic algorithms are often used with machine learning, it would seem reasonable that the machine learning portion of such algorithms could use this KLD as a statistic for the class of problem being solved.

We confined our attention to a particular construction of KLD. Other constructions to the best of the our knowledge have not been explored.

B. Simulated ECGA

We found that look-ahead simulations were effective under the parameters of the original algorithm that we used. It remains to be seen whether such an approach is beneficial under different parameter settings of the original algorithm. For that matter, it remains to be seen whether look-ahead simulations are applicable to problems with χ-ary alphabets as opposed to binary. In addition, appropriate settings for the look-ahead simulation parameters need to be developed with respect to problem size, building block size, etc. as they have been in [19].

Furthermore, we have shown that look-ahead simulations work for Trap-4 problems. However, it remains to be seen how this approach scales to Trap-k problems for larger values of k. Additionally, we defer analyzing the applicability of look-ahead simulations with other deceptive problems and other classes of problems. For instance, we are skeptical whether look-ahead simulations would be beneficial in a dynamic problem environment.

We have given a hybrid approach to the algorithm in terms of using look-ahead simulations during the first few generations, and then returning to the normal algorithm beyond this point. It maybe be entirely possible that other hybrid approaches are valid here as well. For instance, the order-2 behavior algorithms referred to previously [6] may be applicable approaches in lieu of look-ahead simulations.

References

[1] Shumeet Baluja. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report CMU-CS-94-163, Pittsburgh, PA, January 1994.

[2] K.P. Burnham and D.R. Anderson. *Model Selection and Multimodel Inference: A Practical Information-Theoretic Approach*. Springer New York, 2003.

[3] M. Clergue and P. Collard. Ga-hard functions built by combination of trap functions. In *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on*, volume 1, pages 249–254, May 2002.

[4] Jukka Corander, Patrik Waldmann, and Mikko J. Sillanpää. Bayesian analysis of genetic differentiation between populations. *Genetics*, 163(1):367–374, 2003.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[6] Jeremy S De Bonet, Charles L Isbell, Paul Viola, et al. Mimic: Finding optima by estimating probability densities. *Advances in neural information processing systems*, pages 424–430, 1997.

[7] David E Goldberg, Kalyanmoy Deb, and Jeffrey Horn. Massive multimodality, deception, and genetic algorithms. *Urbana*, 51:61801, 1992.

[8] G. R. Harik, F. G. Lobo, and D. E. Goldberg. The compact genetic algorithm. *IEEE Transactions on Evolutionary Computation*, 3(4):287–297, Nov 1999.

[9] Georges Harik. Linkage learning via probabilistic modeling in the ecga. *Urbana*, 51(61):801, 1999.

[10] Georges R. Harik. Finding multimodal solutions using restricted tournament selection. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 24–31, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[11] Georges R. Harik, Fernando G. Lobo, and Kumara Sastry. *Linkage Learning via Probabilistic Modeling in the Extended Compact Genetic Algorithm (ECGA)*, pages 39–61. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[12] Ekaterina A. Holdener, née Smorodkina, and Daniel Tauritz. Dissertation title: the art of parameter-less evolutionary algorithms 4.0/4.0 gpa. 2005.

[13] S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 03 1951.

[14] Heinz Mühlenbein, Thilo Mahnig, and Alberto Ochoa Rodriguez. Schemata, distributions and graphical models in evolutionary optimization. *Journal of Heuristics*, 5(2):215–247, 1999.

[15] Martin Pelikan. *Hierarchical Bayesian Optimization Algorithm*, pages 105–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[16] José C. Pereira and Fernando G. Lobo. A java implementation of the sga, umda, ecga, and HBOA. *CoRR*, abs/1506.07980, 2015.

[17] R. Santana, P. Larrañaga, and J. A. Lozano. Learning factorizations in estimation of distribution algorithms using affinity propagation. *Evolutionary Computation*, 18(4):515–546, Dec 2010.

[18] Kumara Sastry, Hussein A. Abbass, David E. Goldberg, and D. D. Johnson. Sub-structural niching in estimation of distribution algorithms. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 671–678, New York, NY, USA, 2005. ACM.

[19] Kumara Sastry and David E. Goldberg. On extended compact genetic algorithm. Technical report, GECCO-2000, LATE BREAKING PAPERS, GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE, 2000.

Acknowledgements

José C. Pereira

Under the time constraints of this research, none of this would have been possible without the baseline algorithm given in [16]. This made it very easy to extend the functionality due to the great documentation and use of software engineering design patterns.

Fernando G. Lobo

Due to some confusion in terminology in the literature, our research came to a standstill as unexpected results were found from an incorrect parameter setting. We are very grateful to Dr. Lobo in his responsiveness and aid in providing a copy of their original repository of the ECGA.

UMN-TC CSE Labs

The computational resources for this research were made possible by the University of Minnesota - Twin Cities' College of Science and Engineering computer labs where roughly 5000 CPU-hours were used.